Recursive n-gram hashing is pairwise independent, at best

Daniel Lemire[a,*], Owen Kaser[b]

[a] LICEF, Université du Québec à Montréal (UQAM), 100 Sherbrooke West, Montreal, QC, H2X 3P2 Canada
[b] Dept. of CSAS, University of New Brunswick, 100 Tucker Park Road, Saint John, NB, Canada

Abstract

Many applications use sequences of n consecutive symbols (n-grams). Hashing these n-grams can be a performance bottleneck. For more speed, recursive hash families compute hash values by updating previous values. We prove that recursive hash families cannot be more than pairwise independent. While hashing by irreducible polynomials is pairwise independent, our implementations either run in time $O(n)$ or use an exponential amount of memory. As a more scalable alternative, we make hashing by cyclic polynomials pairwise independent by ignoring $n-1$ bits. Experimentally, we show that hashing by cyclic polynomials is twice as fast as hashing by irreducible polynomials. We also show that randomized Karp-Rabin hash families are not pairwise independent.

Key words: Rolling Hashing, Rabin-Karp Hashing, Hashing Strings

1. Introduction

An n-gram is a consecutive sequence of n symbols from an alphabet Σ. An n-gram hash function h maps n-grams to numbers in $[0,2^L)$. These functions have several applications from full-text matching [1–3], pattern matching [4], or language models [5–11] to plagiarism detection [12].

To prove that a hashing algorithm must work well, we typically need hash values to satisfy some statistical property. Indeed, a hash function that maps all n-grams to a single integer would not be useful. Yet, a single hash function is deterministic: it maps an n-gram to a single hash value. Thus, we may be able to choose the input data so that the hash values are biased. Therefore, we randomly pick a function from a family $\mathcal{H}$ of functions [13].

Such a family $\mathcal{H}$ is *uniform* (over L-bits) if all hash values are equiprobable. That is, considering h selected uniformly at random from $\mathcal{H}$, we have $P(h(x)=y)=1/2^L$ for all n-grams x and all hash values y. This condition is weak; the family of constant functions ($h(x)=c$) is uniform[1].

*Corresponding author. Tel.: 00+1+514 987-3000 ext. 2835; fax: 00+1+514 843-2160.
Email addresses: lemire@acm.org (Daniel Lemire), o.kaser@computer.org (Owen Kaser)

[1] We omit families uniform over an arbitrary interval $[0,b)$—not of the form $[0,2^L)$. Indeed, several applications [14, 15] require uniformity over L-bits.

Intuitively, we would want that if an adversary knows the hash value of one n-gram, it cannot deduce anything about the hash value of another n-gram. For example, with the family of constant functions, once we know one hash value, we know them all. The family $\mathcal{H}$ is *pairwise independent* if the hash value of n-gram x_1 is independent from the hash value of any other n-gram x_2. That is, we have $P(h(x_1) = y \wedge h(x_2) = z) = P(h(x_1) = y)P(h(x_2) = z) = 1/4^L$ for all n-grams x_1, x_2, and all hash values y, z with $x_1 \neq x_2$. Pairwise independence implies uniformity. We refer to a particular hash function $h \in \mathcal{H}$ as "uniform" or "a pairwise independent hash function" when the family in question can be inferred from the context.

Moreover, the idea of pairwise independence can be generalized: a family of hash functions $\mathcal{H}$ is *k-wise independent* if given distinct $x_1, \ldots, x_k$ and given h selected uniformly at random from $\mathcal{H}$, then $P(h(x_1) = y_1 \wedge \cdots \wedge h(x_k) = y_k) = 1/2^{kL}$. Note that k-wise independence implies $k-1$-wise independence and uniformity. (Fully) independent families are k-wise independent for arbitrarily large k. For applications, non-independent families may fare as well as fully independent families if the entropy of the data source is sufficiently high [16].

A hash function h is *recursive* [17]—or rolling[18]—if there is a function F computing the hash value of the n-gram $x_2 \ldots x_{n+1}$ from the hash value of the preceding n-gram $(x_1 \ldots x_n)$ and the values of x_1 and x_{n+1}. That is, we have

$$h(x_2, \ldots, x_{n+1}) = F(h(x_1, \ldots, x_n), x_1, x_{n+1}).$$

Ideally, we could compute function F in time $O(L)$ and not, for example, in time $O(Ln)$.

The main contributions of this paper are:

- a proof that recursive hashing is no more than pairwise independent (§ 3);
- a proof that randomized Karp-Rabin can be uniform but never pairwise independent (§ 5);
- a proof that hashing by irreducible polynomials is pairwise independent (§ 7);
- a proof that hashing by cyclic polynomials is not even uniform (§ 9);
- a proof that hashing by cyclic polynomials is pairwise independent—after ignoring $n-1$ consecutive bits (§ 10).

We conclude with an experimental section where we show that hashing by cyclic polynomials is faster than hashing by irreducible polynomials. Table 1 summarizes the algorithms presented.

2. Trailing-zero independence

Some randomized algorithms [14, 15] merely require that the number of trailing zeroes be independent. For example, to estimate the number of distinct n-grams in a large document without enumerating them, we merely have to compute maximal numbers of leading zeroes k among hash values [19]. Naïvely, we may estimate that if a hash value with k leading zeroes is found, we have $\approx 2^k$ distinct n-grams. Such

Table 1: A summary of the hashing function presented and their properties. For GENERAL and CYCLIC, we require $L \geq n$. To make CYCLIC pairwise independent, we need to discard some bits—the resulting scheme is not formally recursive. Randomized Karp-Rabin is uniform under some conditions.

name	cost per n-gram	independence	memory use
non-recursive 3-wise (§ 4)	$O(Ln)$	3-wise	$O(nL\|\Sigma\|)$
Randomized Karp-Rabin (§ 5)	$O(L\log L2^{O(\log^* L)})$	uniform	$O(L\|\Sigma\|)$
GENERAL (§ 7)	$O(Ln)$	pairwise	$O(L\|\Sigma\|)$
RAM-Buffered GENERAL (§ 8)	$O(L)$	pairwise	$O(L\|\Sigma\|+L2^n)$
CYCLIC (§ 9)	$O(L+n)$	pairwise (§ 10)	$O((L+n)\|\Sigma\|)$

estimates might be useful because the number of distinct n-grams grows large with n: Shakespeare's First Folio [20] has over 3 million **distinct** 15-grams.

Formally, let $\text{zeros}(x)$ return the number of trailing zeros (0,1,...,L) of x, where $\text{zeros}(0) = L$. We say h is *k-wise trailing-zero independent* if $P(\text{zeros}(h(x_1)) \geq j_1 \wedge \text{zeros}(h(x_2)) \geq j_2 \wedge \ldots \wedge \text{zeros}(h(x_k)) \geq j_k) = 2^{-j_1-j_2-\cdots-j_k}$, for $j_i = 0,1,\ldots,L$.

If h is k-wise independent, it is k-wise trailing-zero independent. The converse is not true. If h is a k-wise independent function, consider $g \circ h$ where g makes zero all bits before the rightmost 1 (e.g., $g(0101100) = 0000100$). Hash $g \circ h$ is k-wise trailing-zero independent but not even uniform (consider that $P(g = 0001) = 8P(g = 1000)$).

3. Recursive hash functions are no more than pairwise independent

Not only are recursive hash functions limited to pairwise independence: they cannot be 3-wise *trailing-zero* independent.

Proposition 1 *There is no 3-wise trailing-zero independent hashing function that is recursive.*

PROOF Consider the $(n+2)$-gram $\texttt{a}^n\texttt{bb}$. Suppose h is recursive and 3-wise trailing-zero independent, then

$$\begin{aligned}
& P\Big(\text{zeros}(h(\texttt{a},\ldots,\texttt{a})) \geq L \bigwedge \\
& \qquad \text{zeros}(h(\texttt{a},\ldots,\texttt{a},\texttt{b})) \geq L \bigwedge \text{zeros}(h(\texttt{a},\ldots,\texttt{a},\texttt{b},\texttt{b})) \geq L\Big) \\
= \;& P\Big(h(\texttt{a},\ldots,\texttt{a}) = 0 \bigwedge F(0,\texttt{a},\texttt{b}) = 0 \bigwedge F(0,\texttt{a},\texttt{b}) = 0\Big) \\
= \;& P\Big(h(\texttt{a},\ldots,\texttt{a}) = 0 \bigwedge F(0,\texttt{a},\texttt{b}) = 0\Big) \\
= \;& P\Big(\text{zeros}(h(\texttt{a},\ldots,\texttt{a})) \geq L \bigwedge \text{zeros}(h(\texttt{a},\ldots,\texttt{a},\texttt{b})) \geq L\Big) \\
= \;& 2^{-2L} \text{ by trailing-zero pairwise independence} \\
\neq \;& 2^{-3L} \text{ as required by trailing-zero 3-wise independence.}
\end{aligned}$$

Hence, we have a contradiction and no such h exists.

4. A non-recursive 3-wise independent hash function

A trivial way to generate an independent hash is to assign a random integer in $[0, 2^L)$ to each new value x. Unfortunately, this requires as much processing and storage as a complete indexing of all values.

However, in a multidimensional setting this approach can be put to good use. Suppose that we have tuples in $K_1 \times K_2 \times \cdots \times K_n$ such that $|K_i|$ is small for all i. We can construct independent hash functions $h_i : K_i \to [0, 2^L)$ for all i and combine them. The hash function $h(x_1, x_2, \ldots, x_n) = h_1(x_1) \oplus h_2(x_2) \oplus \cdots \oplus h_n(x_n)$ is then 3-wise independent ($\oplus$ is the "exclusive or" function, XOR). In time $O(\sum_{i=1}^{n} |K_i|)$, we can construct the hash function by generating $\sum_{i=1}^{n} |K_i|$ random numbers and storing them in a look-up table. With constant-time look-up, hashing an n-gram thus takes $O(Ln)$ time. Algorithm 1 is an application of this idea to n-grams.

Algorithm 1 The (non-recursive) 3-wise independent family.

Require: n L-bit hash functions $h_1, h_1, \ldots, h_n$ over Σ from an independent hash family

```
1: s ← empty FIFO structure
2: for each character c do
3:     append c to s
4:     if length(s)= n then
5:         yield h_1(s_1) ⊕ h_2(s_2) ⊕ ... ⊕ h_n(s_n)
           {The yield statement returns the value, without terminating the algorithm.}
6:         remove oldest character from s
7:     end if
8: end for
```

This new family is not 4-wise independent for $n > 1$. Consider the n-grams `ac`,`ad`, `bc`, `bd`. The XOR of their four hash values is zero. However, the family is 3-wise independent.

Proposition 2 *The family of hash functions $h(x) = h_1(x_1) \oplus h_2(x_2) \oplus \ldots \oplus h_n(x_n)$, where the L-bit hash functions $h_1, \ldots, h_n$ are taken from an independent hash family, is 3-wise independent.*

PROOF Consider any 3 distinct n-grams: $x^{(1)} = x_1^{(1)} \ldots x_n^{(1)}$, $x^{(2)} = x_1^{(2)} \ldots x_n^{(2)}$, and $x^{(3)} = x_1^{(3)} \ldots x_n^{(3)}$. Because the n-grams are distinct, at least one of two possibilities holds:

Case A For some $i \in \{1, \ldots, n\}$, the three values $x_i^{(1)}, x_i^{(2)}, x_i^{(3)}$ are distinct. Write $\chi_j = h_i(x_i^{(j)})$ for $j = 1, 2, 3$. For example, consider the three 1-grams: `a`,`b`,`c`.

Case B (Up to a reordering of the three n-grams.) There are two values $i, j \in \{1, \ldots, n\}$ such that $x_i^{(1)}$ is distinct from the two identical values $x_i^{(2)}, x_i^{(3)}$, and such that $x_j^{(2)}$ is distinct from the two identical values $x_i^{(1)}, x_i^{(3)}$. Write $\chi_1 = h_i(x_i^{(1)})$, $\chi_2 = h_j(x_j^{(2)})$, and $\chi_3 = h_i(x_i^{(3)})$. For example, consider the three 2-grams: `ad`,`bc`,`bd`.

Recall that the XOR operation is invertible: $a \oplus b = c$ if and only if $a = b \oplus c$.
We prove 3-wise independence for cases A and B.

Case A. Write $f^{(i)} = h(x^{(i)}) \oplus \chi_i$ for $i = 1,2,3$. We have that the values χ_1, χ_2, χ_3 are mutually independent, and they are independent from the values $f^{(1)}, f^{(2)}, f^{(3)}$[2]:

$$P\left(\bigwedge_{i=1}^{3} \chi_i = y_i \wedge \bigwedge_{i=1}^{3} f^{(i)} = y'_i\right) = \prod_{i=1}^{3} P(\chi_i = y_i) P\left(\bigwedge_{i=1}^{3} f^{(i)} = y'_i\right)$$

for all values y_i, y'_i. Hence, we have

$$\begin{aligned}
& P\left(h(x^{(1)}) = z^{(1)} \bigwedge h(x^{(2)}) = z^{(2)} \bigwedge h(x^{(3)}) = z^{(3)}\right) \\
&= P\left(\chi_1 = z^{(1)} \oplus f^{(1)}) \bigwedge \chi_2 = z^{(2)} \oplus f^{(2)} \bigwedge \chi_3 = z^{(3)} \oplus f^{(3)}\right) \\
&= \sum_{\eta,\eta',\eta''} P\left(\chi_1 = z^{(1)} \oplus \eta \bigwedge \chi_2 = z^{(2)} \oplus \eta' \bigwedge \chi_3 = z^{(3)} \oplus \eta''\right) \times \\
&\quad P(f^{(1)} = \eta \wedge f^{(2)} = \eta' \wedge f^{(3)} = \eta'') \\
&= \sum_{\eta,\eta',\eta''} \frac{1}{2^{3L}} P(f^{(1)} = \eta \wedge f^{(2)} = \eta' \wedge f^{(3)} = \eta'') \\
&= \frac{1}{2^{3L}}.
\end{aligned}$$

Thus, in this case, the hash values are 3-wise independent.

Case B. Write $f^{(1)} = h(x^{(1)}) \oplus \chi_1$, $f^{(2)} = h(x^{(2)}) \oplus \chi_2 \oplus \chi_3$, $f^{(3)} = h(x^{(3)}) \oplus \chi_3$. Again, the values χ_1, χ_2, χ_3 are mutually independent, and independent from the values $f^{(1)}, f^{(2)}, f^{(3)}$. We have

$$\begin{aligned}
& P\left(h(x^{(1)}) = z^{(1)} \bigwedge h(x^{(2)}) = z^{(2)} \bigwedge h(x^{(3)}) = z^{(3)}\right) \\
&= P\left(\chi_1 = z^{(1)} \oplus f^{(1)}) \bigwedge \chi_2 \oplus \chi_3 = z^{(2)} \oplus f^{(2)} \bigwedge \chi_3 = z^{(3)} \oplus f^{(3)}\right) \\
&= P\left(\chi_1 = z^{(1)} \oplus f^{(1)}) \bigwedge \chi_2 = z^{(2)} \oplus f^{(2)} \oplus z^{(3)} \oplus f^{(3)} \bigwedge \chi_3 = z^{(3)} \oplus f^{(3)}\right) \\
&= \sum_{\eta,\eta',\eta''} P\left(\chi_1 = z^{(1)} \oplus \eta \bigwedge \chi_2 = z^{(2)} \oplus z^{(3)} \oplus \eta' \oplus \eta'' \bigwedge \chi_3 = z^{(3)} \oplus \eta''\right) \times \\
&\quad P(f^{(1)} = \eta \wedge f^{(2)} = \eta' \wedge f^{(3)} = \eta'') \\
&= \sum_{\eta,\eta',\eta''} \frac{1}{2^{3L}} P(f^{(1)} = \eta \wedge f^{(2)} = \eta' \wedge f^{(3)} = \eta'') \\
&= \frac{1}{2^{3L}}.
\end{aligned}$$

[2]The values $f^{(1)}, f^{(2)}, f^{(3)}$ are not necessarily mutually independent.

This concludes the proof.

5. Randomized Karp-Rabin is not independent

One of the most common recursive hash functions is commonly associated with the Karp-Rabin string-matching algorithm [21]. Given an integer B, the hash value over the sequence of integers $x_1, x_2, \ldots, x_n$ is $\sum_{i=1}^{n} x_i B^{n-i}$. A variation of the Karp-Rabin hash method is "Hashing by Power-of-2 Integer Division" [17], where $h(x_1, \ldots, x_n) = \sum_{i=1}^{n} x_i B^{n-i} \bmod 2^L$. In particular, the `hashcode` method of the Java String class uses this approach, with $L = 32$ and $B = 31$ [22]. A widely used textbook [23, p. 157] recommends a similar Integer-Division hash function for strings with $B = 37$.

Since such Integer-Division hash functions are recursive, quickly computed, and widely used, it is interesting to seek a randomized version of them. Assume that h_1 is a random hash function over symbols uniform in $[0, 2^L)$, then define $h(x_1, \ldots, x_n) = B^{n-1}h_1(x_1) + B^{n-2}h_1(x_2) + \cdots + h_1(x_n) \bmod 2^L$ for some fixed integer B. We choose $B = 37$ (calling the resulting randomized hash "ID37;" see Algorithm 2). Our algorithm computes each hash value in time $O(M(L))$, where $M(L)$ is the cost of multiplying two L-bit integers. (We precompute the value $B^n \bmod 2^L$.) In many practical cases, L bits can fit into a single machine word and the cost of multiplication can be considered constant. In general, $M(L)$ is in $O(L \log L 2^{O(\log^* L)})$ [24].

Algorithm 2 The recursive ID37 family (Randomized Karp-Rabin).

Require: an L-bit hash function h_1 over Σ from an independent hash family

```
1:  B ← 37
2:  s ← empty FIFO structure
3:  x ← 0 (L-bit integer)
4:  z ← 0 (L-bit integer)
5:  for each character c do
6:     append c to s
7:     x ← Bx − B^n z + h_1(c) mod 2^L
8:     if length(s)= n then
9:        yield x
10:       remove oldest character y from s
11:       z ← h_1(y)
12:    end if
13: end for
```

The randomized Integer-Division functions mapping n-grams to $[0, 2^L)$ are not pairwise independent. However, for B odd, they are uniform.

Proposition 3 *Randomized Integer-Division hashing with B odd is not uniform for n-grams, if n is even. Otherwise, it is uniform, but not pairwise independent.*

PROOF For B odd, we see that $P(h(\mathtt{a}^{2k}) = 0) > 2^{-L}$ since $h(\mathtt{a}^{2k}) = h_1(\mathtt{a})(B^0(1+B) + B^2(1+B) + \cdots + B^{2k-2}(1+B)) \bmod 2^L$ and since $(1+B)$ is even, we have $P(h(\mathtt{a}^{2k}) =$

$0) \geq P(h_1(x_1) = 2^{L-1} \vee h_1(x_1) = 0) = 1/2^{L-1}$. Hence, for B odd and n even, we do not have uniformity.

For the rest of the result, we begin with $n = 2$ and B even. If $x_1 \neq x_2$, then $P(h(x_1,x_2) = y) = P(Bh_1(x_1) + h_1(x_2) = y \bmod 2^L) = \sum_z P(h_1(x_2) = y - Bz \bmod 2^L)P(h_1(x_1) = z) = \sum_z P(h_1(x_2) = y - Bz \bmod 2^L)/2^L = 1/2^L$, whereas $P(h(x_1,x_1) = y) = P((B+1)h_1(x_1) = y \bmod 2^L) = 1/2^L$ since $(B+1)x = y \bmod 2^L$ has a unique solution x when B is even. Therefore h is uniform. This argument can be extended for any value of n and for n odd, B even.

To show it is not pairwise independent, first suppose that B is odd. For any string β of length $n-2$, consider n-grams $w_1 = \beta\texttt{aa}$ and $w_2 = \beta\texttt{bb}$ for distinct $\texttt{a},\texttt{b} \in \Sigma$. Then $P(h(w_1) = h(w_2)) = P(B^2h(\beta) + Bh_1(\texttt{a}) + h_1(\texttt{a}) = B^2h(\beta) + Bh_1(\texttt{b}) + h_1(\texttt{b}) \bmod 2^L) = P((1+B)(h_1(\texttt{a}) - h_1(\texttt{b})) \bmod 2^L = 0) \geq P(h_1(\texttt{a}) - h_1(\texttt{b}) = 0) + P(h_1(\texttt{a}) - h_1(\texttt{b}) = 2^{L-1})$. Because h_1 is independent, $P(h_1(\texttt{a}) - h_1(\texttt{b}) = 0) = \sum_{c \in [0,2^L)} P(h_1(\texttt{a}) = c)P(h_1(\texttt{b}) = c) = \sum_{c \in [0,2^L)} 1/4^L = 1/2^L$. Moreover, $P(h_1(\texttt{a}) - h_1(\texttt{b}) = 2^{L-1}) > 0$. Thus, we have that $P(h(w_1) = h(w_2)) > 1/2^L$ which contradicts pairwise independence. Second, if B is even, a similar argument shows $P(h(w_3) = h(w_4)) > 1/2^L$, where $w_3 = \beta\texttt{aa}$ and $w_4 = \beta\texttt{ba}$. $P(h(\texttt{a},\texttt{a}) = h(\texttt{b},\texttt{a})) = P(Bh_1(\texttt{a}) + h_1(\texttt{a}) = Bh_1(\texttt{b}) + h_1(\texttt{a}) \bmod 2^L) = P(B(h_1(\texttt{a}) - h_1(\texttt{b})) \bmod 2^L = 0) \geq P(h_1(\texttt{a}) - h_1(\texttt{b}) = 0) + P(h_1(\texttt{a}) - h_1(\texttt{b}) = 2^{L-1}) > 1/2^L$. This argument can be extended for any value of B and n.

A weaker condition than pairwise independence is 2-universality: a family is 2-universal if $P(h(x_1) = h(x_2)) \leq 1/2^L$ [16]. As a consequence of this proof, Randomized Integer-Division is not even 2-universal.

These results also hold for any Integer-Division hash where the modulo is by an even number, not necessarily a power of 2.

6. Generating hash families from polynomials over Galois fields

A practical form of hashing using the binary Galois field GF(2) is called "Recursive Hashing by Polynomials" and has been attributed to Kubina by Cohen [17]. GF(2) contains only two values (1 and 0) with the addition (and hence subtraction) defined by XOR, $a+b = a \oplus b$ and the multiplication by AND, $a \times b = a \wedge b$. GF(2)[x] is the vector space of all polynomials with coefficients from GF(2). Any integer in binary form (e.g., $c = 1101$) can thus be interpreted as an element of GF(2)[x] (e.g., $c = x^3 + x^2 + 1$). If $p(x) \in$ GF(2)[x], then GF(2)[x]/$p(x)$ can be thought of as GF(2)[x] modulo $p(x)$. As an example, if $p(x) = x^2$, then GF(2)[x]/$p(x)$ is the set of all linear polynomials. For instance, $x^3 + x^2 + x + 1 = x + 1 \bmod x^2$ since, in GF(2)[x], $(x+1) + x^2(x+1) = x^3 + x^2 + x + 1$.

As a summary, we compute operations over GF(2)[x]/$p(x)$—where $p(x)$ is of degree L—as follows:

- the polynomial $\sum_{i=0}^{L-1} q_i x^i$ is represented as the L-bit integer $\sum_{i=0}^{L-1} q_i 2^i$;
- subtraction or addition of two polynomials is the XOR of their L-bit integers;

Table 2: Some irreducible polynomials over GF(2)[x]

degree	polynomial
10	$1+x^3+x^{10}$
15	$1+x+x^{15}$
20	$1+x^3+x^{20}$
25	$1+x^3+x^{25}$
30	$1+x+x^4+x^6+x^{30}$

- multiplication of a polynomial $\sum_{i=0}^{L} q_i x^i$ by the monomial x is represented either as $\sum_{i=0}^{L-1} q_i x^{i+1}$ if $q_{L-1}=0$ or as $p(x)+\sum_{i=0}^{L-1} q_i x^{i+1}$ otherwise. In other words, if the value of the last bit is 1, we merely apply a binary left shift, otherwise, we apply a binary left shift immediately followed by an XOR with the integer representing $p(x)$. In either case, we get an L-bit integer.

Hence, merely with the XOR operation, the binary left shift, and a way to evaluate the value of the last bit, we can compute all necessary operations over GF(2)[x]$/p(x)$ using integers.

Consider a hash function h_1 over characters taken from some independent family. Interpreting h_1 hash values as polynomials in GF(2)[x]$/p(x)$, and with the condition that degree$(p(x)) \geq n$, we define a hash function as $h(a_1,a_2,\cdots,a_n) = h_1(a_1)x^{n-1} + h_1(a_2)x^{n-2}+\cdots+h_1(a_n)$. It *is* recursive over the sequence $h_1(a_i)$. The combined hash can be computed by reusing previous hash values:

$$h(a_2,a_3,\ldots,a_{n+1}) = xh(a_1,a_2,\ldots,a_n) - h_1(a_1)x^n + h_1(a_{n+1}).$$

Depending on the choice of the polynomial $p(x)$ we get different hashing schemes, including GENERAL and CYCLIC, which are presented in the next two sections.

7. Recursive hashing by irreducible polynomials is pairwise independent

We can choose $p(x)$ to be an irreducible polynomial of degree L in GF(2)[x]: an irreducible polynomial cannot be factored into nontrivial polynomials (see Table 2). The resulting hash is called GENERAL (see Algorithm 3). The main benefit of setting $p(x)$ to be an irreducible polynomial is that GF(2)[x]$/p(x)$ is a field; in particular, it is impossible that $p_1(x)p_2(x) = 0 \bmod p(x)$ unless either $p_1(x)=0$ or $p_2(x)=0$. The field property allows us to prove that the hash function is pairwise independent.

Lemma 1 GENERAL *is pairwise independent.*

PROOF If $p(x)$ is irreducible, then any non-zero $q(x) \in$ GF(2)[x]$/p(x)$ has an inverse, noted $q^{-1}(x)$ since GF(2)[x]$/p(x)$ is a field. Interpret hash values as polynomials in GF(2)[x]$/p(x)$.

Firstly, we prove that GENERAL is uniform. In fact, we show a stronger result: $P(q_1(x)h_1(a_1)+q_2(x)h_1(a_2)+\cdots+q_n(x)h_1(a_n)=y) = 1/2^L$ for any polynomials q_i where at least one is different from zero. The result follows by induction

Algorithm 3 The recursive GENERAL family.

Require: an L-bit hash function h_1 over Σ from an independent hash family; an irreducible polynomial p of degree L in GF$(2)[x]$

$s \leftarrow$ empty FIFO structure
$x \leftarrow 0$ (L-bit integer)
$z \leftarrow 0$ (L-bit integer)
for each character c **do**
  append c to s
  $x \leftarrow \text{shift}(x)$
  $z \leftarrow \text{shift}^n(z)$
  $x \leftarrow x \oplus z \oplus h_1(c)$
  if length(s)$= n$ **then**
    yield x
    remove oldest character y from s
    $z \leftarrow h_1(y)$
  end if
end for

function shift
input L-bit integer x
shift x left by 1 bit, storing result in an $L+1$-bit integer x'
if leftmost bit of x' is 1 **then**
  $x' \leftarrow x' \oplus p$
end if
{leftmost bit of x' is thus always 0}
return rightmost L bits of x'

on the number of non-zero polynomials: it is clearly true where there is a single non-zero polynomial $q_i(x)$, since $q_i(x)h_1(a_i) = y \iff q_i^{-1}(x)q_i(x)h_1(a_i) = q_i^{-1}(x)y$. Suppose it is true up to $k-1$ non-zero polynomials and consider a case where we have k non-zero polynomials. Assume without loss of generality that $q_1(x) \neq 0$, we have $P(q_1(x)h_1(a_1) + q_2(x)h_1(a_2) + \cdots + q_n(x)h_1(a_n) = y) = P(h_1(a_1) = q_1^{-1}(x)(y - q_2(x)h_1(a_2) - \cdots - q_n(x)h_1(a_n))) = \sum_{y'} P(h_1(a_1) = q_1^{-1}(x)(y-y'))P(q_2(x)h_1(a_2) + \cdots + q_n(x)h_1(a_n) = y') = \sum_{y'} \frac{1}{2^L}\frac{1}{2^L} = \frac{1}{2^L}$ by the induction argument. Hence the uniformity result is shown.

Consider two distinct sequences $a_1, a_2, \ldots, a_n$ and $a'_1, a'_2, \ldots, a'_n$. Write $H_a = h(a_1, a_2, \ldots, a_n)$ and $H_{a'} = h(a'_1, a'_2, \ldots, a'_n)$. We have that $P(H_a = y \wedge H_{a'} = y') = P(H_a = y | H_{a'} = y')P(H_{a'} = y')$. Hence, to prove pairwise independence, it suffices to show that $P(H_a = y | H_{a'} = y') = 1/2^L$.

Suppose that $a_i = a'_j$ for some i, j; if not, the result follows since by the (full) independence of the hashing function h_1, the values H_a and $H_{a'}$ are independent. Write $q(x) = -(\sum_{k|a_k=a_i} x^{n-k})(\sum_{k|a'_k=a'_j} x^{n-k})^{-1}$, then $H_a + q(x)H_{a'}$ is independent from $a_i = a'_j$ (and $h_1(a_i) = h_1(a'_j)$).

In $H_a + q(x)H_{a'}$, only hashed values $h_1(a_k)$ for $a_k \neq a_i$ and $h_1(a'_k)$ for $a'_k \neq a'_j$ remain: label them $h_1(b_1), \ldots, h_1(b_m)$. The result of the substitution can be written

$H_a + q(x)H_{a'} = \sum_k q_k(x)h_1(b_k)$ where $q_k(x)$ are polynomials in GF(2)$[x]/p(x)$. All $q_k(x)$ are zero if and only if $H_a + q(x)H_{a'} = 0$ for all values of $h_1(a_1),\ldots,h_1(a_n)$ and $h_1(a'_1),\ldots,h_1(a'_n)$ (but notice that the value $h_1(a_i) = h_1(a'_j)$ is irrelevant); in particular, it must be true when $h_1(a_k) = 1$ and $h_1(a'_k) = 1$ for all k, hence $(x^n + \cdots + x + 1) + q(x)(x^n \ldots + x + 1) = 0 \Rightarrow q(x) = -1$. Thus, all $q_k(x)$ are zero if and only if $H_a = H_{a'}$ for all values of $h_1(a_1),\ldots,h_1(a_n)$ and $h_1(a'_1),\ldots,h_1(a'_n)$ which only happens if the sequences a and a' are identical. Hence, not all $q_k(x)$ are zero.

Write $H_{y',a'} = (\sum_{k|a'_k=a'_j} x^{n-k})^{-1}(y' - \sum_{k|a'_k\neq a'_j} x^{n-k}h_1(a'_k))$. On the one hand, the condition $H_{a'} = y'$ can be rewritten as $h_1(a'_j) = H_{y',a'}$. On the other hand, $H_a + q(x)H_{a'} = y + q(x)y'$ is independent from $h_1(a'_j) = h_1(a_i)$. Because $P(h_1(a'_j) = H_{y',a'}) = 1/2^L$ irrespective of y' and $h_1(a'_k)$ for $k \in \{k|a'_k \neq a'_j\}$, then $P(h_1(a'_j) = H_{y',a'}|H_a + q(x)H_{a'} = y + q(x)y') = P(h_1(a'_j) = H_{y',a'})$ which implies that $h_1(a'_j) = H_{y',a'}$ and $H_a + q(x)H_{a'} = y + q(x)y'$ are independent. Hence, we have

$$\begin{aligned} & P(H_a = y|H_{a'} = y') \\ = \;& P(H_a + q(x)H_{a'} = y + q(x)y'|h_1(a'_j) = H_{y',a'}) \\ = \;& P(H_a + q(x)H_{a'} = y + q(x)y') \\ = \;& P(\sum_k q_k(x)h_1(b_k) = y + q(x)y') \end{aligned}$$

and by the earlier uniformity result, this last probability is equal to $1/2^L$. This concludes the proof.

8. Trading memory for speed: RAM-Buffered GENERAL

Unfortunately, GENERAL—as computed by Algorithm 3—requires $O(nL)$ time per n-gram. Indeed, shifting a value n times in GF(2)$[x]/p(x)$ requires $O(nL)$ time. However, if we are willing to trade memory usage for speed, we can precompute these shifts. We call the resulting scheme RAM-Buffered GENERAL.

Lemma 2 *Pick any $p(x)$ in GF(2)$[x]$. The degree of $p(x)$ is L. Represent elements of GF(2)$[x]/p(x)$ as polynomials of degree at most $L-1$. Given any h in GF(2)$[x]/p(x)$. we can compute $x^n h$ in $O(L)$ time given an $O(L2^n)$-bit memory buffer.*

PROOF Write h as $\sum_{i=0}^{L-1} q_i x^i$. Divide h into two parts, $h^{(1)} = \sum_{i=0}^{L-n-1} q_i x^i$ and $h^{(2)} = \sum_{i=L-n}^{L-1} q_i x^i$, so that $h = h^{(1)} + h^{(2)}$. Then $x^n h = x^n h^{(1)} + x^n h^{(2)}$. The first part, $x^n h^{(1)}$ is a polynomial of degree at most $L-1$ since the degree of $h^{(1)}$ is at most $L-1-n$. Hence, $x^n h^{(1)}$ as an L-bit value is just $q_{L-n-1}q_{L-n-2}\ldots q_0 0\ldots 0$. which can be computed in time $O(L)$. So, only the computation of $x^n h^{(2)}$ is possibly more expensive than $O(L)$ time, but $h^{(2)}$ has only n terms as a polynomial (since the first $L-n$ terms are always zero). Hence, if we precompute $x^n h^{(2)}$ for all 2^n possible values of $h^{(2)}$, and store them in an array with $O(L)$ time look-ups, we can compute $x^n h$ as an L-bit value in $O(L)$ time.

When n is large, this precomputation requires excessive space and precomputation time. Fortunately, we can trade back some speed for memory. Consider the proof of

Lemma 2. Instead of precomputing the shifts of all 2^n possible values of $h^{(2)}$ using an array of 2^n entries, we can further divide $h^{(2)}$ into K parts. For simplicity, assume that the integer K divides n. The K parts $h^{(2,1)},\ldots,h^{(2,K)}$ are made of the first n/K bits, the next n/K bits and so on. Because $x^n h^{(2)} = \sum_{i=1}^{K} x^n h^{(2,i)}$, we can shift $h^{(2)}$ by n in $O(KL)$ operations using K arrays of $2^{n/K}$ entries. To summarize, we have a time complexity of $O(KL)$ per n-gram using $O(L|\Sigma| + LK2^{n/K})$ bits. We implemented the case $K = 2$.

9. Recursive hashing by cyclic polynomials is not even uniform

Choosing $p(x) = x^L + 1$ for $L \geq n$, for any polynomial $q(x) = \sum_{i=0}^{L-1} q_i x^i$, we have

$$x^i q(x) = x^i(q_{L-1}x^{L-1} + \cdots + q_1 x + q_0) = q_{L-i-1}x^{L-i-2} + \cdots + q_{L-i+1}x + q_{L-i}.$$

Thus, we have that multiplication by x^i is a bitwise rotation, a cyclic left shift—which can be computed in $O(L)$ time. The resulting hash (see Algorithm 4) is called CYCLIC. It requires only $O(L)$ time per hash value. *Empirically*, Cohen showed that CYCLIC is uniform [17]. In contrast, we show that it is not formally uniform:

Algorithm 4 The recursive CYCLIC family.

Require: an L-bit hash function h_1 over Σ from an independent hash family

```
s ← empty FIFO structure
x ← 0 (L-bit integer)
z ← 0 (L-bit integer)
for each character c do
    append c to s
    rotate x left by 1 bit
    rotate z left by n bits
    x ← x ⊕ z ⊕ h_1(c)
    if length(s)= n then
        yield x
        remove oldest character y from s
        z ← h_1(y)
    end if
end for
```

Lemma 3 CYCLIC *is not uniform for n even and never 2-universal, and thus never pairwise independent.*

PROOF If n is even, use the fact that $x^{n-1} + \cdots + x + 1$ is divisible by $x + 1$ to write $x^{n-1} + \cdots + x + 1 = (x+1)r(x)$ for some polynomial $r(x)$. Clearly, $r(x)(x+1)(x^{L-1} + x^{L-2} + \cdots + x + 1) = 0 \bmod x^L + 1$ for any $r(x)$ and so $P(h(a_1,a_1,\ldots,a_1) = 0) = P((x^{n-1} + \cdots + x + 1)h_1(a_1) = 0) = P((x+1)r(x)h_1(a_1) = 0) \geq P(h_1(a_1) = 0 \vee h_1(a_1) = x^{L-1} + x^{L-2} + \cdots + x + 1) = 1/2^{L-1}$. Therefore, CYCLIC is not uniform for n even.

To show CYCLIC is never pairwise independent, consider $n = 3$ (for simplicity), then $P(h(a_1,a_1,a_2) = h(a_1,a_2,a_1)) = P((x+1)(h_1(a_1) + h_1(a_2)) = 0) \geq P(h_1(a_1) +$

Table 3: CYCLIC hash for various values of $h_1(\mathtt{a})$ ($h(\mathtt{a},\mathtt{a}) = xh_1(\mathtt{a}) + h_1(\mathtt{a}) \bmod 2^L + 1$)

$h_1(\mathtt{a})$	$h(\mathtt{a},\mathtt{a})$	$h(\mathtt{a},\mathtt{a})$ (first two bits)	$h(\mathtt{a},\mathtt{a})$ (last two bits)	$h(\mathtt{a},\mathtt{a})$ (first and last bit)
000	000	00	00	00
100	110	11	10	10
010	011	01	11	01
110	101	10	01	11
001	101	10	01	11
101	011	01	11	01
011	110	11	10	10
111	000	00	00	00

$h_1(a_2) = 0 \vee h_1(a_1) + h_1(a_2) = x^{L-1} + x^{L-2} + \cdots + x + 1) = 1/2^{L-1}$, but 2-universal hash values are equal with probability $1/2^L$. The result is shown.

Of the four recursive hashing functions investigated by Cohen [17], GENERAL and CYCLIC were superior both in terms of speed and uniformity, though CYCLIC had a small edge over GENERAL. For n large, the benefits of these recursive hash functions compared to the 3-wise independent hash function presented earlier can be substantial: n table look-ups is much more expensive than a single look-up followed by binary shifts.

10. CYCLIC is pairwise independent if you remove $n-1$ consecutive bits

Because Cohen found empirically that CYCLIC had good uniformity [17], it is reasonable to expect CYCLIC to be *almost* uniform and maybe even *almost* pairwise independent. To illustrate this intuition, consider Table 3 which shows that while $h(\mathtt{a},\mathtt{a})$ is not uniform ($h(\mathtt{a},\mathtt{a}) = 001$ is impossible), $h(\mathtt{a},\mathtt{a})$ minus any bit is indeed uniformly distributed. We will prove that this result holds in general.

The next lemma and the next theorem show that CYCLIC is quasi-pairwise independent in the sense that $L-n+1$ consecutive bits (e.g., the first or last $L-n+1$ bits) are pairwise independent. In other words, CYCLIC is pairwise independent if we are willing to sacrifice $n-1$ bits. (We say that n bits are "consecutive modulo L" if the bits are located at indexes $i \bmod L$ for n consecutive values of i such as $i = k, k+1, \ldots, k+n-1$.)

Lemma 4 *If $q(x) \in GF(2)[x]/(x^L+1)$ (with $q(x) \neq 0$) has degree $n < L$, then*

- *the equation $q(x)w = y \bmod x^L + 1$ modulo the first n bits[3] has exactly 2^n solutions for all y;*

[3]By "equality modulo ⟨*some specified set of bit positions*⟩", we mean that the two quantities are bitwise identical, with exceptions permitted only at the specified positions. For our polynomials, "equality modulo the first n bit positions" implies the difference of the two polynomials has degree at most $n-1$.

- *more generally, the equation* $q(x)w = y \bmod x^L + 1$ *modulo any consecutive* n *bits (modulo* L*) has exactly* 2^n *solutions for all* y*.*

PROOF Let P be the set of polynomials of degree at most $L-n-1$. Take any $p(x) \in P$, then $q(x)p(x)$ has degree at most $L-n-1+n = L-1$ and thus if $q(x) \neq 0$ and $p(x) \neq 0$, then $q(x)p(x) \neq 0 \bmod x^L+1$. Hence, for any distinct $p_1, p_2 \in P$ we have $q(x)p_1 \neq q(x)p_2 \bmod x^L+1$.

To prove the first item, we begin by showing that there is always exactly one solution in P. Consider that there are 2^{L-n} polynomials $p(x)$ in P, and that all values $q(x)p(x)$ are distinct. Suppose there are $p_1, p_2 \in P$ such that $q(x)p_1 = q(x)p_2 \bmod x^L+1$ modulo the first n bits, then $q(x)(p_1 - p_2)$ is a polynomial of degree at most $n-1$ while $p_1 - p_2$ is a polynomial of degree at most $L-n-1$ and $q(x)$ is a polynomial of degree n, thus $p_1 - p_2 = 0$. (If $p1 - p2 \neq 0$ then $\text{degree}(q(x)(p1-p2) \bmod x^L+1) \geq \text{degree}(q(x)) = n$, a contradiction.) Hence, all $p(x)$ in P are mapped to distinct values modulo the first n bits, and since there are 2^{L-n} such distinct values, the result is shown.

Any polynomial of degree $L-1$ can be decomposed into the form $p(x) + x^{L-n}z(x)$ where $z(x)$ is a polynomial of degree at most $n-1$ and $p(x) \in P$. By the preceding result, for distinct $p_1, p_2 \in P$, $q(x)(x^{L-n}z(x) + p_1)$ and $q(x)(x^{L-n}z(x) + p_2)$ must be distinct modulo the first n bits. In other words, the equation $q(x)(x^{L-n}z(x) + p) = y$ modulo the first n bits has exactly one solution $p \in P$ for any $z(x)$ and since there are 2^n polynomials $z(x)$ of degree at most $n-1$, then $q(x)w = y$ (modulo the first n bits) must have 2^n solutions.

To prove the second item, choose j and use the first item to find any w solving $q(x)w = yx^j \bmod x^L+1$ modulo the first n bits. j. Then wx^{L-j} is a solution to $q(x)w = y \bmod x^L+1$ modulo the bits in positions $j, j+1, \ldots, j+n-1 \bmod L$.

We have the following corollary to Lemma 4.

Corollary 1 *If* w *is chosen uniformly at random in* $GF(2)[x]/(x^L+1)$*, then* $P(q(x)w = y \bmod n-1$ *bits*$) = 1/2^{L-n+1}$ *where the* $n-1$ *bits are consecutive (modulo* L*).*

Theorem 1 *Consider the* L*-bit* CYCLIC *n-gram hash family. Pick any* $n-1$ *consecutive bit locations, then remove these bits from all hash values. The resulting* $L-n+1$*-bit hash family is pairwise independent.*

PROOF We show $P(q_1(x)h_1(a_1) + q_2(x)h_1(a_2) + \cdots + q_n(x)h_1(a_n) = y \bmod n-1 \text{ bits}) = 1/2^{L-n+1}$ for any polynomials q_i where at least one is different from zero. It is true when there is a single non-zero polynomial $q_i(x)$ by Corollary 1. Suppose it is true up to $k-1$ non-zero polynomials and consider a case where we have k non-zero polynomials. Assume without loss of generality that $q_1(x) \neq 0$, we have $P(q_1(x)h_1(a_1) + q_2(x)h_1(a_2) + \cdots + q_n(x)h_1(a_n) = y \bmod n-1 \text{ bits}) = P(q_1(x)h_1(a_1) = y - q_2(x)h_1(a_2) - \cdots - q_n(x)h_1(a_n) \bmod n-1 \text{ bits}) = \sum_{y'} P(q_1(x)h_1(a_1) = y - y' \bmod n-1 \text{ bits})P(q_2(x)h_1(a_2) + \cdots + q_n(x)h_1(a_n) = y' \bmod n-1 \text{ bits}) = \sum_{y'} \frac{1}{2^{L-n+1}} \frac{1}{2^{L-n+1}} = 1/2^{L-n+1}$ by the induction argument, where the sum is over 2^{L-n+1} values of y'. Hence the uniformity result is shown.

Consider two distinct sequences $a_1, a_2, \ldots, a_n$ and $a'_1, a'_2, \ldots, a'_n$. Write $H_a = h(a_1, a_2, \ldots, a_n)$ and $H_{a'} = h(a'_1, a'_2, \ldots, a'_n)$. To prove pairwise independence, it suffices to show that $P(H_a = y \bmod n-1 \text{ bits} | H_{a'} = y' \bmod n-1 \text{ bits}) = 1/2^{L-n+1}$. Suppose that $a_i = a'_j$ for some i, j; if not, the result follows by the (full) independence of the hashing function h_1. Using Lemma 4, find $q(x)$ such that $q(x)\sum_{k|a'_k=a'_j} x^{n-k} = -\sum_{k|a_k=a_i} x^{n-k} \bmod n-1$ bits, then $H_a + q(x)H_{a'} \bmod n-1$ bits is independent from $a_i = a'_j$ (and $h_1(a_i) = h_1(a'_j)$).

The hashed values $h_1(a_k)$ for $a_k \neq a_i$ and $h_1(a'_k)$ for $a'_k \neq a'_j$ are now relabelled as $h_1(b_1), \ldots, h_1(b_m)$. Write $H_a + q(x)H_{a'} = \sum_k q_k(x)h_1(b_k) \bmod n-1$ bits where $q_k(x)$ are polynomials in $\mathrm{GF}(2)[x]/(x^L+1)$ (not all $q_k(x)$ are zero). As in the proof of Lemma 1, we have that $H_{a'} = y' \bmod n-1$ bits and $H_a + q(x)H_{a'} = y + q(x)y' \bmod n-1$ bits are independent[4]: $P(H_{a'} = y' \bmod n-1 \text{ bits} | y', b_1, b_2, \ldots, b_m) = 1/2^{L-n+1}$ by Corollary 1 since $H_{a'} = y$ can be written as $r(x)h_1(a'_j) = y - \sum_k r_k(x)h_1(b_k)$ for some polynomials $r(x), r_1(x), \ldots, r_m(x)$. Hence, we have

$$\begin{aligned}
&P(H_a = y \bmod n-1 \text{ bits} | H_{a'} = y' \bmod n-1 \text{ bits}) \\
&\quad = \quad P(H_a + q(x)H_{a'} = y + q(x)y' \bmod n-1 \text{ bits} | H_{a'} = y' \bmod n-1 \text{ bits}) \\
&\quad = \quad P(H_a + q(x)H_{a'} = y + q(x)y' \bmod n-1 \text{ bits}) \\
&\quad = \quad P(\sum_k q_k(x)h_1(b_k) = y + q(x)y' \bmod n-1 \text{ bits})
\end{aligned}$$

and by the earlier uniformity result, this last probability is equal to $1/2^{L-n+1}$.

11. Experimental comparison

Irrespective of $p(x)$, computing hash values has complexity $\Omega(L)$. For GENERAL and CYCLIC, we require $L \geq n$. Hence, the computation of their hash values is in $\Omega(n)$. For moderate values of L and n, this analysis is pessimistic because CPUs can process 32- or 64-bit words in one operation.

To assess their real-world performance, the various hashing algorithms[5] were written in C++. We compiled them with the GNU GCC 4.0.1 compiler on an Apple MacBook with two Intel Core 2 Duo processors (2.4 GHz) and 4 GiB of RAM. The -O3 compiler flag was used since it provided slightly better performance for all algorithms. All hash values are stored using 32-bit integers, irrespective of the number of bits used.

All hashing functions generate 19-bit hash values, except for CYCLIC which generates 19+n-bit hash values. We had CYCLIC generate more bits to compensate for the fact that it is only pairwise independent after removal of $n-1$ consecutive bits. For GENERAL, we used the polynomial $p(x) = x^{19} + x^{18} + x^{17} + x^{16} + x^{12} + x^7 + x^6 + x^5 + x^3 + x^2 + 1$ [25]. For Randomized Karp-Rabin, we used the ID37 family. The character hash-values are stored in an array for fast look-up.

[4] We use the shorthand notation $P(f(x,y) = c | x, y) = b$ to mean $P(f(x,y) = c | x = z_1, y = z_2) = b$ for all values of z_1, z_2.

[5] `http://code.google.com/p/ngramhashing/`.

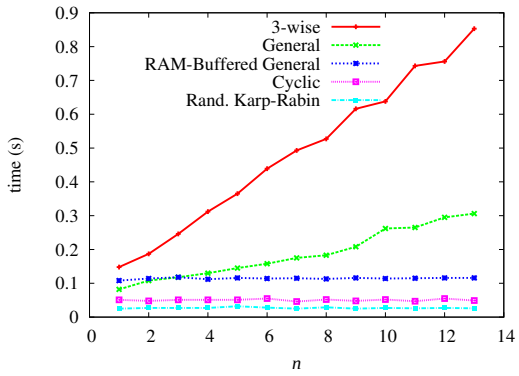

Figure 1: Wall-clock running time to hash all n-grams in the King James Bible

We report wall-clock time in Fig. 1 for hashing the n-grams of the King James Bible [20] which contains 4.3 million ASCII characters. CYCLIC is twice as fast as GENERAL. As expected, the running time of the non-recursive hash function (3-wise) grows linearly with n: for $n = 5$, 3-wise is already seven times slower than CYCLIC. Speed-wise, Randomized Karp-Rabin (ID37) is the clear winner, being nearly twice as fast as CYCLIC. The performance of CYCLIC and ID37 is oblivious to n in this test.

The RAM-Buffered GENERAL timings are—as expected—independent of n, but they are twice as large as the CYCLIC timings. We do not show the modified version of RAM-Buffered GENERAL that uses two precomputed arrays instead of a single one. It was approximately 30% slower than ordinary RAM-Buffered GENERAL, even up to $n = 25$. However, its RAM usage was 3 orders of magnitude smaller: from 135 MB down to 25 kB. Overall, we cannot recommend RAM-Buffered GENERAL or its modification considering that (1) its memory usage grows as 2^n and (2) it is slower than CYCLIC.

12. Conclusion

Considering speed and pairwise independence, we recommend CYCLIC—after discarding $n-1$ consecutive bits. If we require only uniformity, Randomized Integer-Division is twice as fast.

Acknowledgments

This work is supported by NSERC grants 155967, 261437 and by FQRNT grant 112381. The authors are grateful to the anonymous reviewers for their significant contributions.

References

[1] J. D. Cohen, Hardware-assisted algorithm for full-text large-dictionary string matching using n-gram hashing, Information Processing and Management 34 (4) (1998) 443–464.

[2] J. D. Cohen, Massive query resolution for rapid selective dissemination of information, Journal of the American Society for Information Science 50 (3) (1999) 195–206.

[3] J. D. Cohen, An n-gram hash and skip algorithm for finding large numbers of keywords in continuous text streams, Softw. Pract. Exper. 28 (15) (1998) 1605–1635.

[4] T. Tan, S. Gould, D. Williams, E. Peltzer, R. Barrie, Fast pattern matching using large compressed databases, US Patent App. 11/326,131 (2006).

[5] H. Schwenk, Continuous space language models, Computer Speech & Language 21 (3) (2007) 492–518.

[6] X. Li, Y. Zhao, A fast and memory-efficient N-gram language model lookup method for large vocabulary continuous speech recognition, Computer Speech & Language 21 (1) (2007) 1–25.

[7] A. Cardenal-Lopez, F. J. Diguez-Tirado, C. Garcia-Mateo, Fast LM look-ahead for large vocabulary continuous speech recognition using perfect hashing, in: ICASSP'02, 2002, pp. 705–708.

[8] X. Zhang, Y. Zhao, Minimum perfect hashing for fast N-gram language model lookup, in: Seventh International Conference on Spoken Language Processing, ISCA, 2002, pp. 401–404.

[9] D. Talbot, M. Osborne, Smoothed Bloom filter language models: Tera-scale LMs on the cheap, in: EMNLP'07, 2007, pp. 468–476.

[10] D. Talbot, M. Osborne, Randomised language modelling for statistical machine translation, in: ACL'07, 2007, pp. 512–519.

[11] D. Talbot, T. Brants, Randomized language models via perfect hash functions, ACL'08 (2008) 505–513.

[12] R. L. Ribler, M. Abrams, Using visualization to detect plagiarism in computer science classes, in: INFOVIS'00, IEEE Computer Society, Washington, DC, USA, 2000, p. 173.

[13] L. Carter, M. N. Wegman, Universal classes of hash functions, Journal of Computer and System Sciences 18 (2) (1979) 143–154.

[14] P. Flajolet, G. N. Martin, Probabilistic counting algorithms for data base applications, Journal of Computer and System Sciences 31 (2) (1985) 182–209.

[15] P. B. Gibbons, S. Tirthapura, Estimating simple functions on the union of data streams, in: SPAA'01, 2001, pp. 281–291.

[16] M. Mitzenmacher, S. Vadhan, Why simple hash functions work: exploiting the entropy in a data stream, in: SODA '08, 2008, pp. 746–755.

[17] J. D. Cohen, Recursive hashing functions for n-grams, ACM Trans. Inf. Syst. 15 (3) (1997) 291–320.

[18] S. Schleimer, D. S. Wilkerson, A. Aiken, Winnowing: local algorithms for document fingerprinting, in: SIGMOD'2003, 2003, pp. 76–85.

[19] M. Durand, P. Flajolet, Loglog counting of large cardinalities, in: ESA'03, Vol. 2832 of LNCS, 2003, pp. 605–617.

[20] Project Gutenberg Literary Archive Foundation, Project Gutenberg, `http://www.gutenberg.org/` (checked 2009-08-03) (2009).

[21] R. M. Karp, M. O. Rabin, Efficient randomized pattern-matching algorithms, IBM Journal of Research and Development 31 (2) (1987) 249–260.

[22] Sun Microsystems, String (Java 2 Platform SE 5.0), online documentation: `http://java.sun.com/j2se/1.5.0/docs/api/index.html` (2004).

[23] M. Weiss, Data Structures and Algorithm Analysis in Java, Addison Wesley, 1999.

[24] M. Fürer, Faster integer multiplication, in: STOC '07, 2007, pp. 57–66.

[25] F. Ruskey, The (combinatorial) object server, `http://www.theory.cs.uvic.ca/~cos/cos.html`, checked 2007-05-30 (2006).